Uniprocessor Scheduling

1 Introduction

As we know, multiprogramming is used to improve the efficiency of the micro-processor. That is when the currently running process is blocked due to waiting for an I/O event, another process may be scheduled to run. At this moment, the issue of *scheduling* is involved, deciding which process should be executed by the processor, because usually more than one process may be ready for being scheduled.

1.1 Types of processor scheduling

There are three different types of scheduling:

- **Long-term scheduling** is performed to decide if a new process is to be created and be added to the pool of processes.

  Long-term scheduling controls the degree of multiprogramming. The more processes that are created, the smaller is the percentage of time that each process can be executed. Thus the long term scheduler may limit the degree of multiprogramming to provide satisfactory service to the current set of processes. Whenever a process terminates, or the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked.

  What process to be created is another issue that long-term scheduling needs to deal with. The decision may be made on a first-come-first-served basis or it can be a tool to manage system performance. For example, if the information is available, the scheduler may attempt to keep a mix of processor-bound and I/O-bound processes. A *processor-bound process* is one that mainly performs computational work and occasionally uses I/O devices, while a *I/O-bound process* is one that uses I/O devices more than the micro-processor.

- **Medium-term scheduling** is a part of the swapping function. It decides if a process should be loaded into the main memory either completely or partially so as to be available for execution. The swapping mechanism has been discussed in previous chapters.

- **Short-term scheduling** is the most common use of the term *scheduling*, i.e. deciding which ready process to execute next. The short-term scheduler, also known as the dispatcher, is invoked whenever an event occurs that may lead to the suspension of the current process or that may provide an opportunity to preempt a currently running process in favor of another. Examples of such events include:
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals

Short-term scheduling is the major concern of this chapter.

Combined with the state transition of processes, the relationships among the above three types of scheduling may be illustrated in Figure 1.

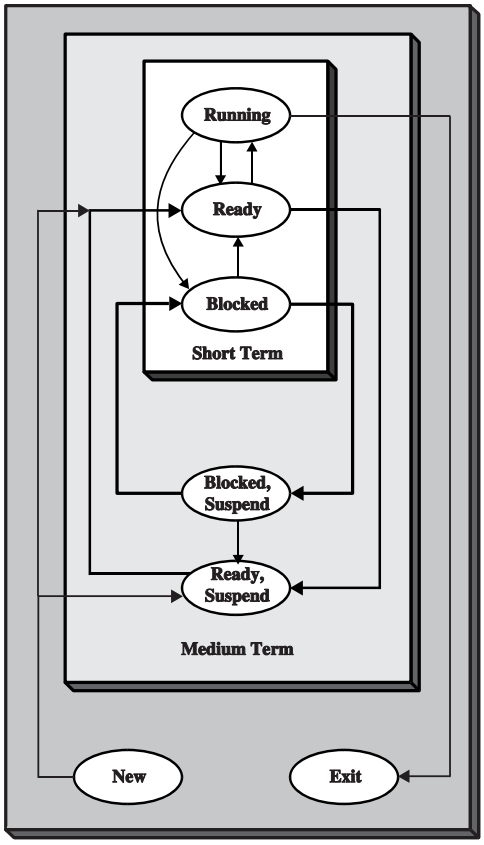


Figure 1: Levels of scheduling

Figure 1 associates the three levels of scheduling with different storage facilities, the outmost storage devices accommodating file systems for executable programs, the virtual memory space for blocked processes, and main memory for processes that may be executed right away.

1.2 Criteria

There are various algorithms available for the short-term scheduling work. We need to know how good they are. That is some criteria are needed to evaluate the performance of those algorithms.

The major criteria relating to processor scheduling are as follows:

- **Turnaround time** is the interval of time between the submission of a process and its completion. This is an appropriate measure for a process in a batch operating system.
- **Response time** is the elapsed time between the submission of a request until the response appears.
- **Throughput** is the rate at which processes are completed. The scheduling policy should attempt to maximize the throughput so that more tasks could be performed.
- **Processor utilization** is the percentage time that the processor is busy. For a shared system, this is a significant criterion, while in single-user systems and real-time systems, this criterion is less important than some of others.
- **Fairness** addresses if some processes suffer starvation. Fairness should be enforced in most systems.

These criteria may be categorized into two groups: user-oriented and system-oriented. The former focuses on the properties that are visible and of interest to the users. For example, in an interactive system, a user always wishes to get response as soon as possible. This may be measured by *response time*. Some criteria are system oriented, focusing on effective and efficient utilization of the processor, such as throughout.

System-oriented criteria are usually important on multi-user operating systems, while on the single-user system, it is probably not important to achieve high processor utilization or high throughput as long as the single user's need is fully met.

It is obvious that the above criteria are interdependent and cannot be optimized simultaneously. For example, providing good response time may require a scheduling algorithm that switches between processes frequently, which increases the overhead of the system, reducing throughput.

In a particular operating system, some criteria may be of more importance than others, thus the designer of the operating system may simply focus on improving those concerned aspects.

1.3 Priority

In many systems, a process is assigned a *priority* and the scheduler will always choose a process of higher priority over one of lower priority. Figure 2 depicts the revised process queue model with the consideration of priority.

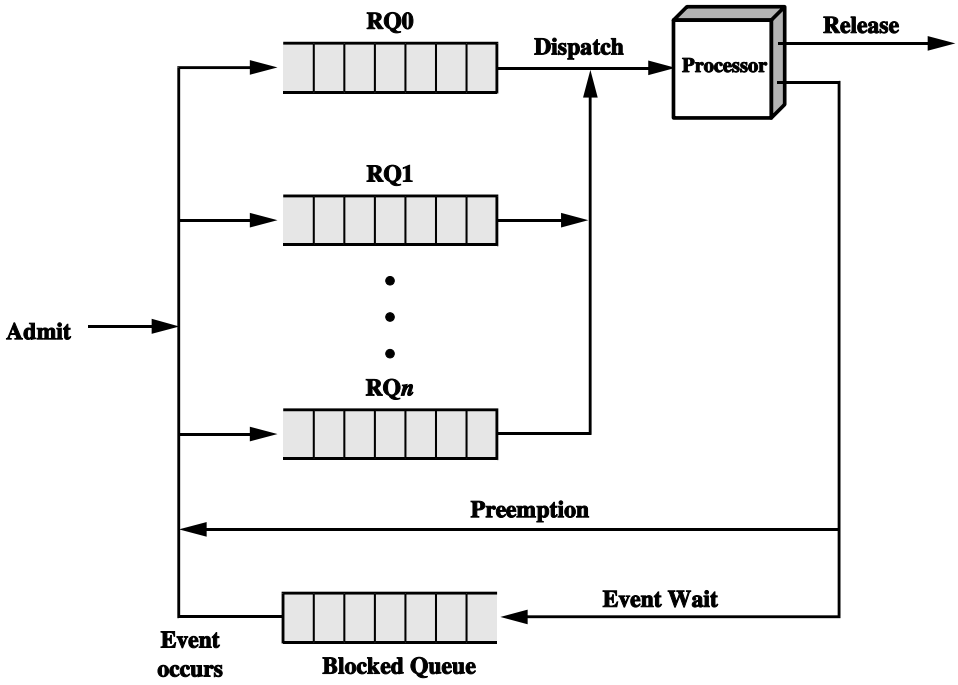


Figure 2: Priority queuing

The new model splits the READY queue into several separate queues, each associated with a specific priority. The dispatcher always chooses a process in the queue with highest priority to execute, and if the queue is empty, then the one with lower priority next to it will be used, and so on.

However a problem that such a system has to face is that a process with low priority may suffer starvation. If there are always processes with higher priority ready, that process will not get any chance to be scheduled. To solve this problem, the priorities of processes may be adjusted dynamically. For example, a process that has waited for a longer time is assigned a higher priority, so that eventually any process will be dispatched whether it has a higher priority or a lower one.

1.4 Preemption

Another issue relating to scheduling is whether a running process could be *preempted* or not. There are two categories:

- **Nonpreemptive**: In this case, a running process continues to execute until (a) it terminates or (b) blocks itself to wait for I/O or to request some operating system service.
- **Preemptive**: The currently running process may be interrupted and moved to the Ready state by the operating system. The preemption may possibly be made due to the arrival of a new process, or the occurrence of an interrupt that places a blocked process in the READY state.

Preemptive policies incur greater overhead than nonpreemptive ones but may be preferred since they prevent some processes from monopolizing the processor for a long time.

2 Scheduling algorithms

Based on the above discussion, we move on to explore various scheduling policies. To compare the performance of these different strategies, we will use the following process set as an example. The table shows when the processes arrive respectively and how much time they need to get completed.

Process	**Arrival Time**	**Service Time (T_s)**
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Figure 3 shows the results when the various policies are applied to the example.

If we use *turnaround time* to measure the performance of various algorithms, we may obtain Table 1, which also includes the so-called *normalized turnaround time*, which is the ratio of turnaround time to service time. It indicates the relative delay experienced by a process. The minimum possible value for this ratio is of course 1.0; increasing values correspond to a decreasing level of service.

Table 1: A performance comparison of scheduling policies

							Mean
FCFS	Finish Time	3	9	13	18	20	
	Turnaround Time (T_r)	3	7	9	12	12	8.60
	T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$	Finish Time	4	18	17	20	15	
	Turnaround Time (T_r)	4	16	13	14	7	10.80
	T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$	Finish Time	3	17	11	20	15	
	Turnaround Time (T_r)	3	15	7	14	11	10.00
	T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN	Finish Time	3	9	15	20	11	
	Turnaround Time (T_r)	3	7	11	14	3	7.60
	T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	Finish Time	3	15	8	20	10	
	Turnaround Time (T_r)	3	13	4	14	2	7.20
	T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	Finish Time	3	9	13	20	15	
	Turnaround Time (T_r)	3	7	9	14	7	7.20
	T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$	Finish Time	4	20	16	19	11	
	Turnaround Time (T_r)	4	18	12	13	3	10.00
	T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$	Finish Time	4	17	18	20	14	
	Turnaround Time (T_r)	4	15	14	14	6	10.60
	T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

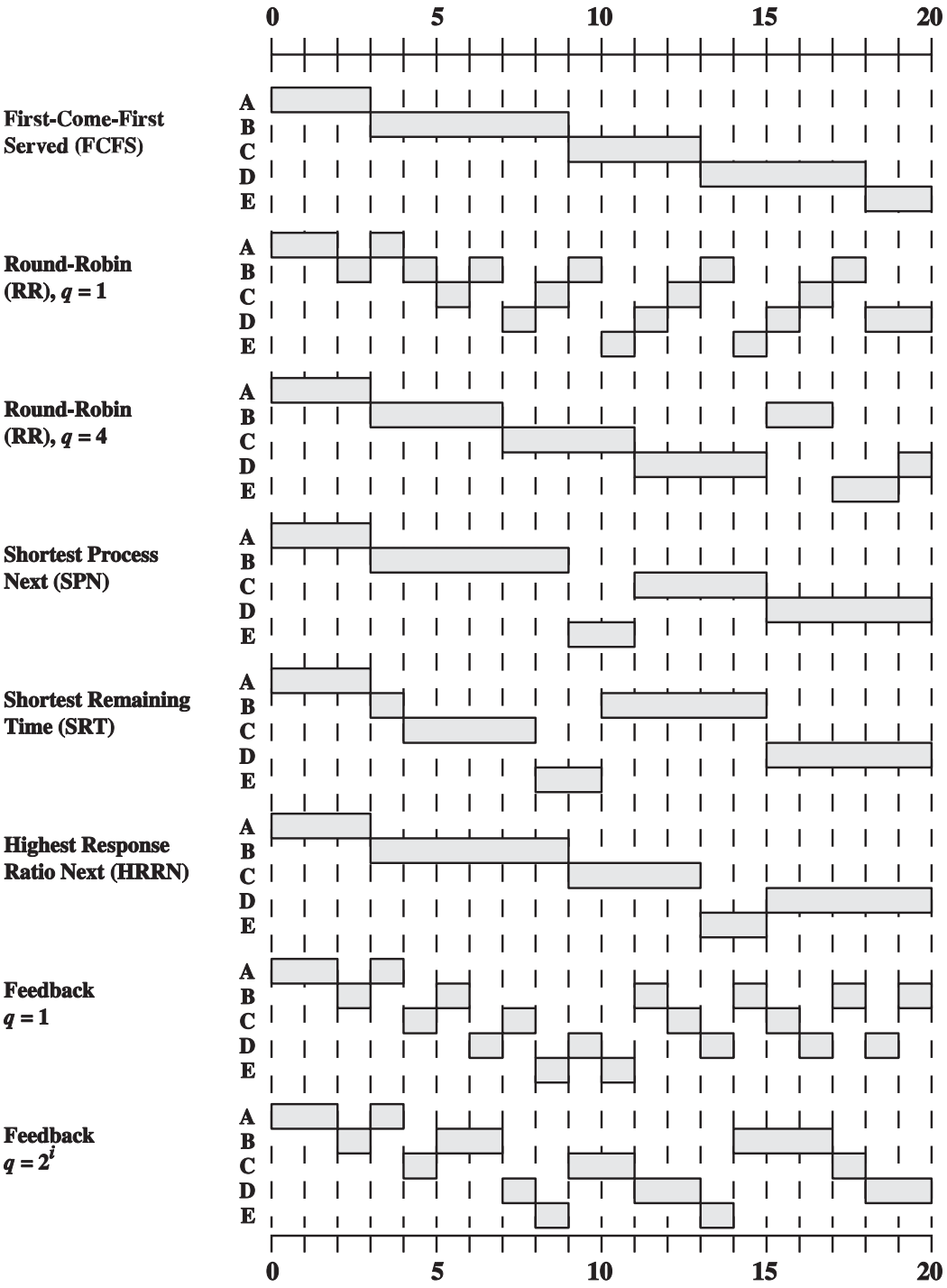


Figure 3: A comparison of scheduling policies

2.1 First-Come-First-Served

First-come-first-served (FCFS) is the simplest scheduling policy, also known as FIFO. With this policy, when a process becomes ready, it joins the ready queue and when the currently running process finishes, the process at the head of the ready queue, which has waited there for the longest time, is selected for execution.

As we can see from Table 1, though process E requires only 2 units of service time, it has to wait until all the preceding processes complete. When we line up for checking out books in the library, we of course are unwilling to be behind some guy who will borrow a hundred books. So FCFS performs much better for long processes and is pretty unfair for short ones.

Thus a system with FCFS tends to favor processor-bound processes over I/O-bound ones, since the latter, though requiring relative light use of the processor, has to wait a long time for the completion of processes before it in the ready queue before it gets dispatched for a short time and is blocked for I/O operation. The same thing may repeatedly happen during the execution of the I/O-bound process. Noticeably, while the process is waiting, the I/O devices that are supposed to be used will be idle, thus leading to the inefficiency of I/O devices.

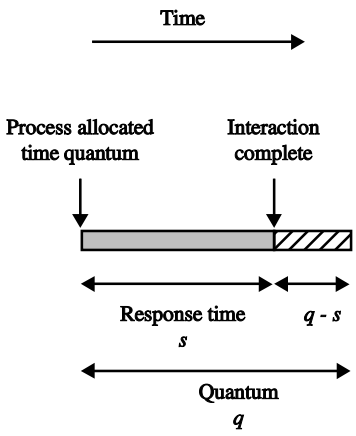
Hence FCFS is not an attractive alternative on its own, but it may be combined with a priority scheme to provide an effective scheduler.

2.2 Round Robin

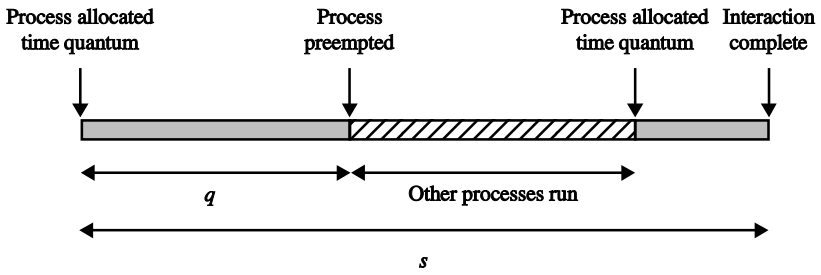
A straightforward way to reduce the suffering of short processes is to use a time-sharing scheme, called *round robin*, with which the operating system assigns short time slices to each process and if the slices allocated for a process are not enough for it to complete, then the process has to wait until its time slice comes again. With the help of a clock, whenever a clock interrupt occurs, the operating system will check if the time slice for the current process ends. If yes, then another process will be scheduled and allocated a time slice.

With round robin, the key design issue is the length of the time slice or quantum. If the slice is short, then short processes tend to get chance to run early. However very short time slices should be avoided, since there is overhead involved in handling the clock interrupt and performing the scheduling. In principle, a time slice should be slightly greater than the time required for a typical interaction. Figure 4 illustrates the effect the decision has on response time. Figure 3 and Table 1 show the results for our example using time quanta q of 1 and 4 time units respectively.

The round robin policy is generally more effective than FCFS, however the I/O-bound processes are still to some extent treated unfairly, because these processes are very likely to be



(a) Time quantum greater than typical interaction



(b) Time quantum less than typical interaction

Figure 4: Effect of size of preemption time quantum

blocked before they use up a complete time quantum, while the processor-bound processes generally make great use of time slices.

An improvement to round robin is that when a blocked process is waked up, it is put into an auxiliary ready queue instead of the regular one. The processes in the auxiliary queue have higher priority than those in the latter, though they are allowed to run up to the length of time slices they are supposed to use. This approach, called *virtual round robin* (VRR), is illustrated in Figure 5. Experiment has shown that VRR is indeed superior to round robin in terms of fairness.

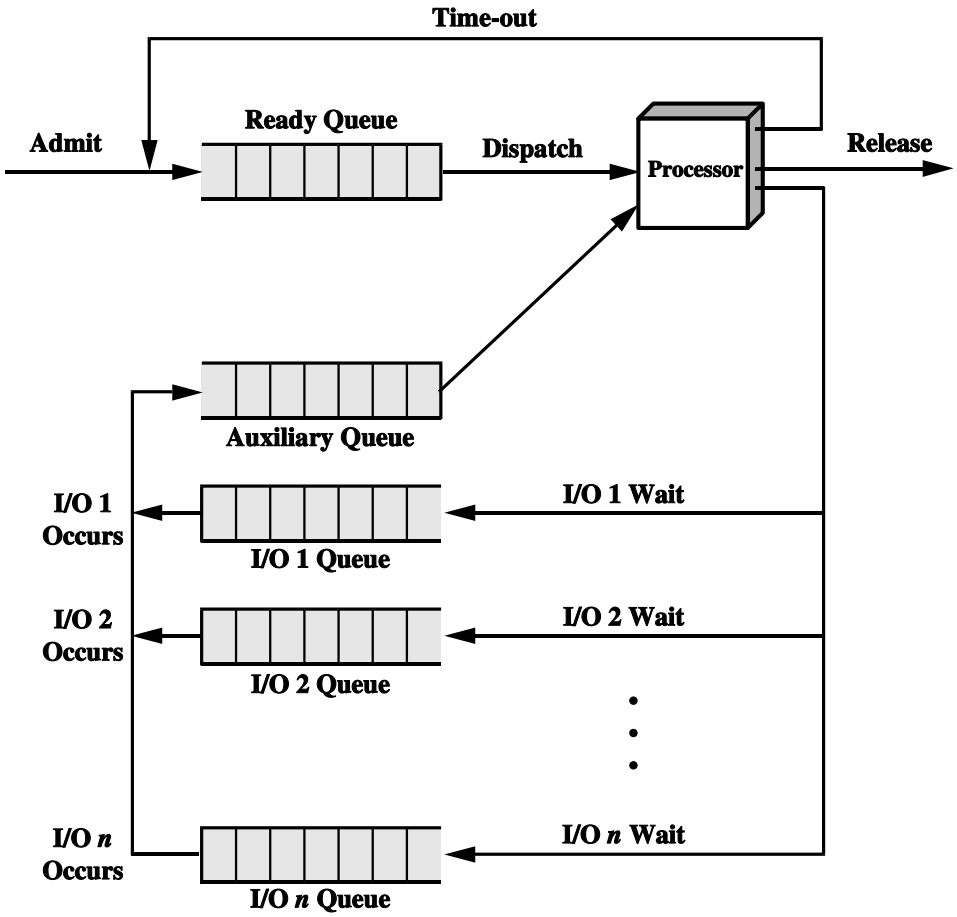


Figure 5: Queuing diagram for virtual round robin scheduler

2.3 Shortest Process Next

Another policy avoiding long processes in favor in FCFS is a nonpreemptive one, called *shortest process next* (SPN), in which the process with the shortest expected execution time is selected next.

Figure 3 and Table 1 show the results for our example.

However one difficulty with the SPN is that we need to know in advance the required processing time of each process. Again, it is impossible for us to know the future, though in batch environment, the programmers may possibly specify the quantum of the processing time of each process. A similar approach as the LRU paging policy in the last chapter may be used to predict the future based on the history. We will not touch the details here, which are included in the text.

The SPN policy is in favor of shorter processes, which however may lead to starvation of longer processes, if shorter ones come in constantly. On the other hand, it is not suitable for a

time-sharing or transaction processing environment because of the lack of preemption.

2.4 Shortest Remaining Time

The shortest remaining time (SRT) policy is a preemptive version of SPN. Whenever a new process is brought in, the scheduler will run to perform the selection function to see if the new process has shorter remaining time than the currently running one. If yes, the current process is preempted and the new one is scheduled.

Compared with SPN, the SRT policy's preemption makes it possible to run a shorter new process. On the other hand, unlike round robin, the SRT policy has less overhead since it does not need to respond to timer interrupts and may simply let the current process run into completion unless there comes in a new short process.

2.5 Highest Response Ratio Next

As we mentioned above, the normalized turnaround time may be used to measure the performance of scheduling algorithms. The highest response ratio next (HRRN) policy is proposed to minimize the average value of the normalized turnaround time over all processes.

For each process in the process pool, we first compute the following ratio:

$$R = \frac{w + s}{s}$$

where w is the time since the process was created and s is the expected service time. Then whenever the current process is blocked or completes, the process with the greatest value will be scheduled to run.

Since a smaller denominator (smaller expected service time) in the fraction results in a greater value, shorter processes are favored. And again as SRT and SPN, the expected service time needs to be estimated based on the history.

2.6 Feedback

Another approach to favor shorter processes is to penalize processes that have been running longer, thus avoiding predicting the expected processing time.

To do so, a dynamic priority mechanism is used. As Figure 6 shows, the ready process queue is split into several queues, each with a different priority. When a process first enters the system, it is put in the queue RQ0, which has the highest priority. After its first execution and

when it becomes READY again, it is placed in RQ1, and so on and so forth. The feedback approach is also preemptive. Like round robin, a specific quantum of time is allocated to each scheduled process. When the time is out, the current process is preempted and another process is chosen from the queues on the highest-priority-first basis. The processes in the same queue follow a FCFS policy. Figure 3 and Table 1 shows our example in the case when the quantum of time is one unit of time.

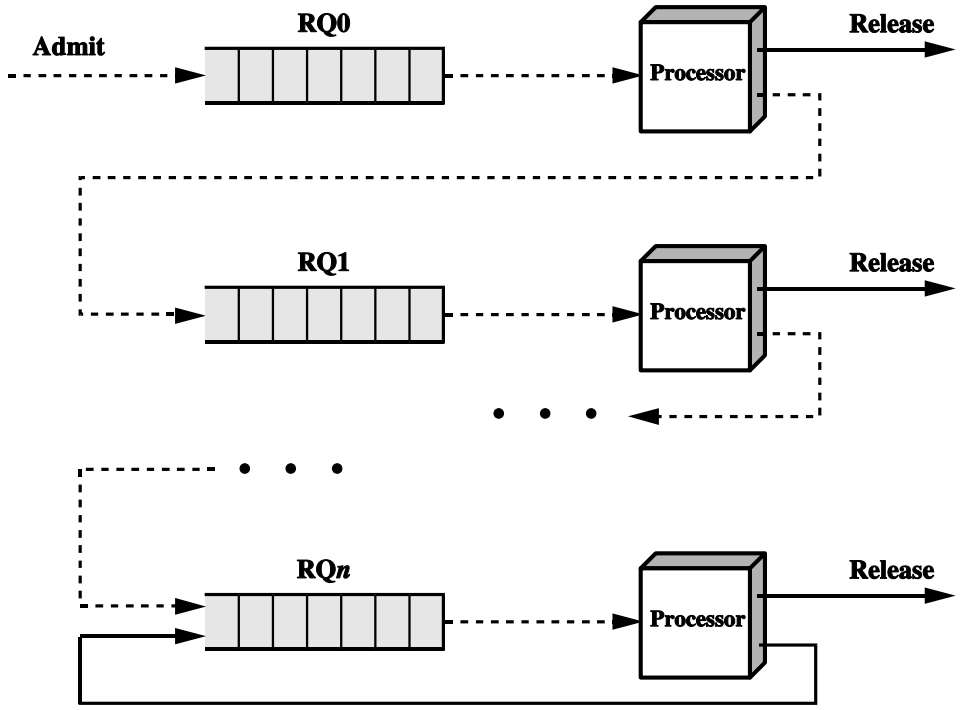


Figure 6: Feedback scheduling

Obviously, shorter processes are favored over longer ones since the latter tend to gradually drift downward and cannot get a chance to run for a long time until there are no processes of higher priority. Starvation is also possible in the feedback policy if new processes come in frequently.